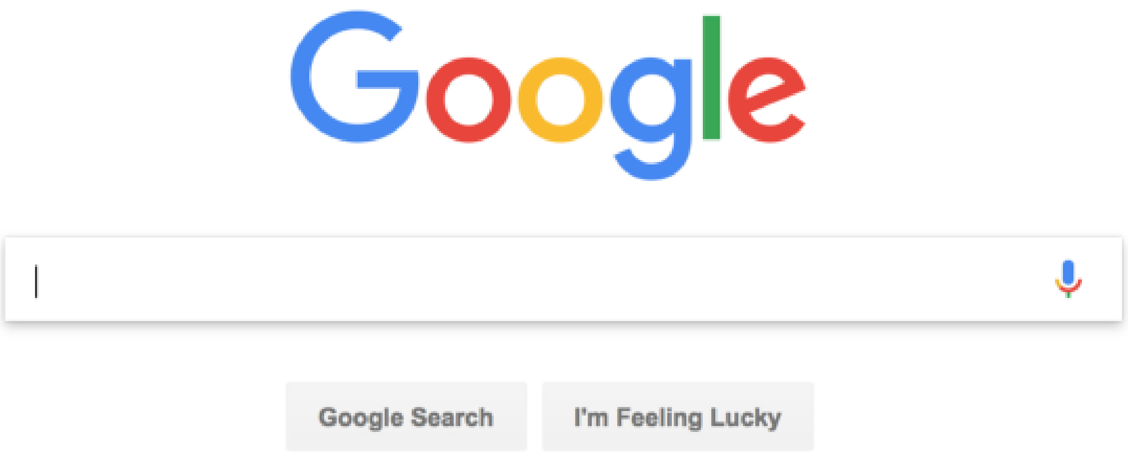
Deangela Neves Follow
Apr 3, 2018 · 7 min read · Listen

Save

How to Build a Search Engine from Scratch in Python — Part 1

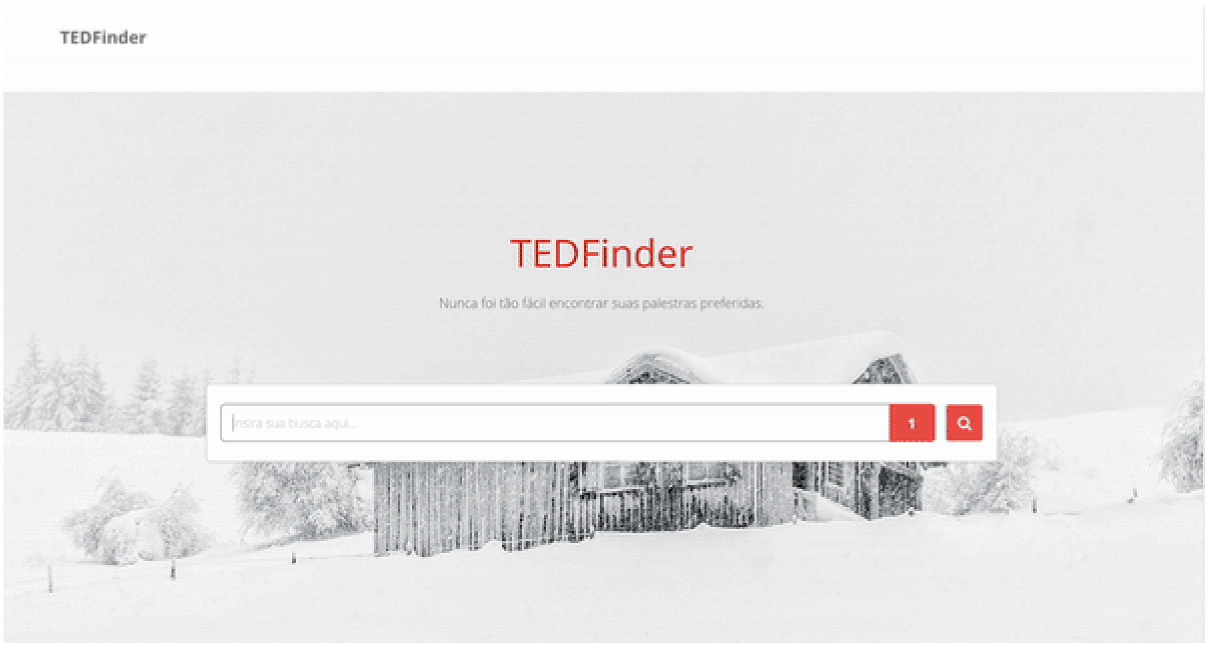


Have you ever wondered how does Google search work? Most of the time, it gives you exactly the results you need based only on a few input words. To understand how it works, let's try to build our own search engine using Python 3.

What is a search engine?

Search engines are basically programs designed to search for items in a database that matches a query given by the user. They can be built to run locally or be web-based applications, and the items searched can be of any type: web pages, images, videos, etc. But the concepts of information retrieval and data mining required to do so are basically the same.

TedFinder, and by the end of this tutorial it should be looking like this:



Set up

We're going to develop it in the following environment:

- Python 3.5.0
- Django 2.0.1
- Scikit-Learn 0.18.1
- Numpy 1.13.3
- NLTK 3.2.5

Download the list of all packages required and install them by running:

```
pip install -r requirements.txt
```

Database

We are going to use a collection of TED talks transcripts as our database. They were

The database is in CSV format and has only two fields:

- **Transcript**- The whole transcript of each talk
- **URL**- The video url corresponding to the talk transcript

Reading data

Create a new Python script and define a simple function to load the database as a Pandas DataFrame:

```python
import pandas as pd

def load_data(path):
  dataframe = pd.read_csv(path)
  return dataframe
```

load_dataset.py hosted with ❤ by GitHub view raw

Feature Extraction

In order to know which talks best fit user's needs, we need to compare the content of a search query to the content of talks somehow. To do that, we are going to use a text mining technique called **TF-IDF**.

TF-IDF was also used by Google in its earliest days and stands for *Term frequency-Inverse Document Frequency.* It is basically a statistic used to evaluate the importance of a word to a document in a collection.In our case, each talk transcript and search query can be seen as a document.

> **Note: If you already know what TF-IDF is and how it works, you can skip to section "All in one"**

Giving weights to words

To better understand these concepts, let's say we have a collection of 5 short documents as shown below.

```
document_3 = "Watching horror movies alone at night is really scary"

document_4 = "He loves to watch films filled with suspense and
unexpected plot twists"

document_5 = "My mom loves to watch movies. My dad hates movie
theaters. My brothers like any kind of movie. And I haven't watched
a single movie since I got into college"

documents = [document_1, document_2, document_3, document_4,
document_5]
```

As we can see, all documents are somehow related to movies. TF-IDF, as its name suggests, weights terms in documents by calculating the product between their frequency and inverse document frequency.

$$weight_i = tf_i * idf_i$$

So, let's see how to calculate these 2 different frequency measures separately.

Preprocessing

To count how many times a word occurs in a document, we need to split it into a list of words first. That's pretty easy to do in Python and we can do that with just one line of code:

```
documents = [document.split(" ") for document in documents]
```

We'll get the following lists of words as result:

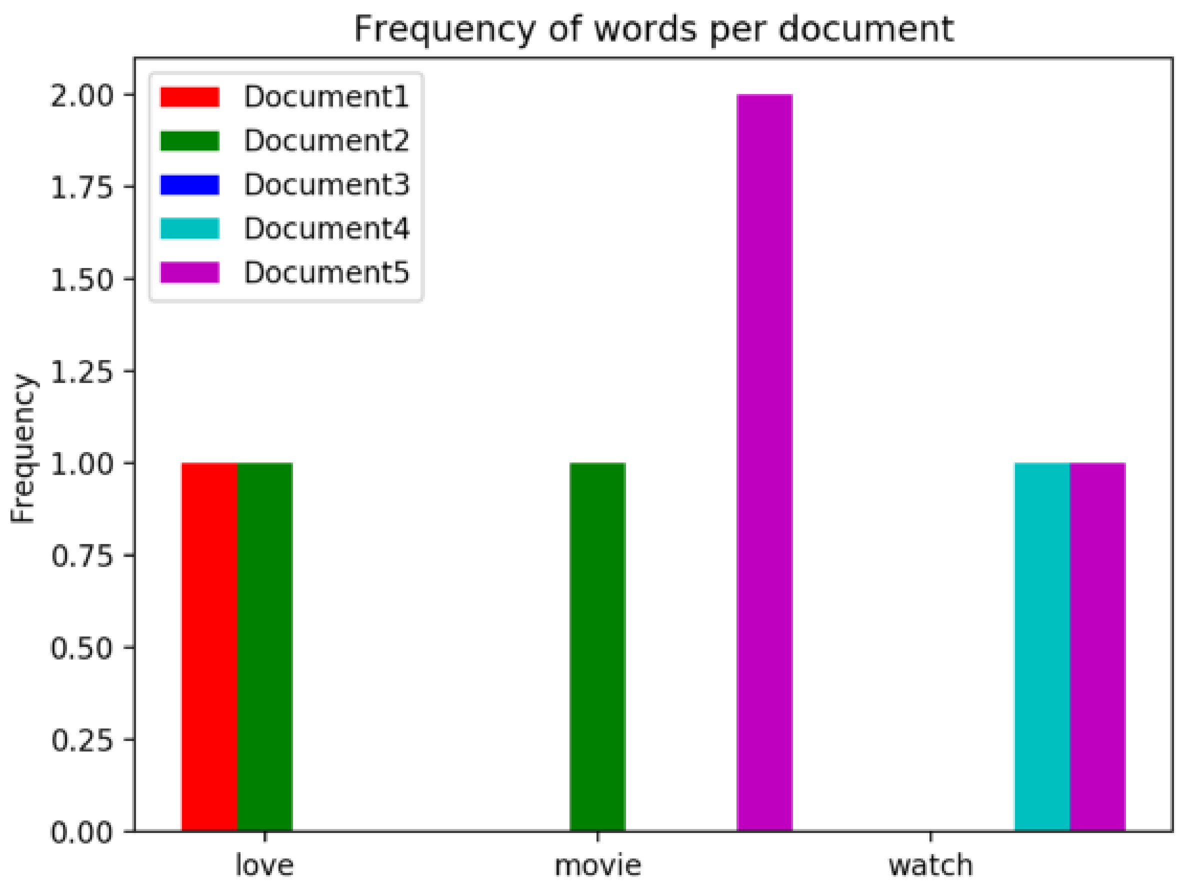
```
document_1 = ['I', 'love', 'watching', 'movies', 'when', "it's",
'cold', 'outside']

document_2 = ['Toy', 'Story', 'is', 'the', 'best', 'animation',
'movie', 'ever,', 'I', 'love', 'it!']

document_3 = ['Watching', 'horror', 'movies', 'alone', 'at',
'night', 'is', 'really', 'scary']

document_4 = ['He', 'loves', 'to', 'watch', 'films', 'filled',
 'a', 'single', 'movie', 'since', 'I', 'got', 'into', 'college']
```

Now, let's define a function to count the repetitions of words per document and another function to plot histo r analyze the distribution of words in each document.

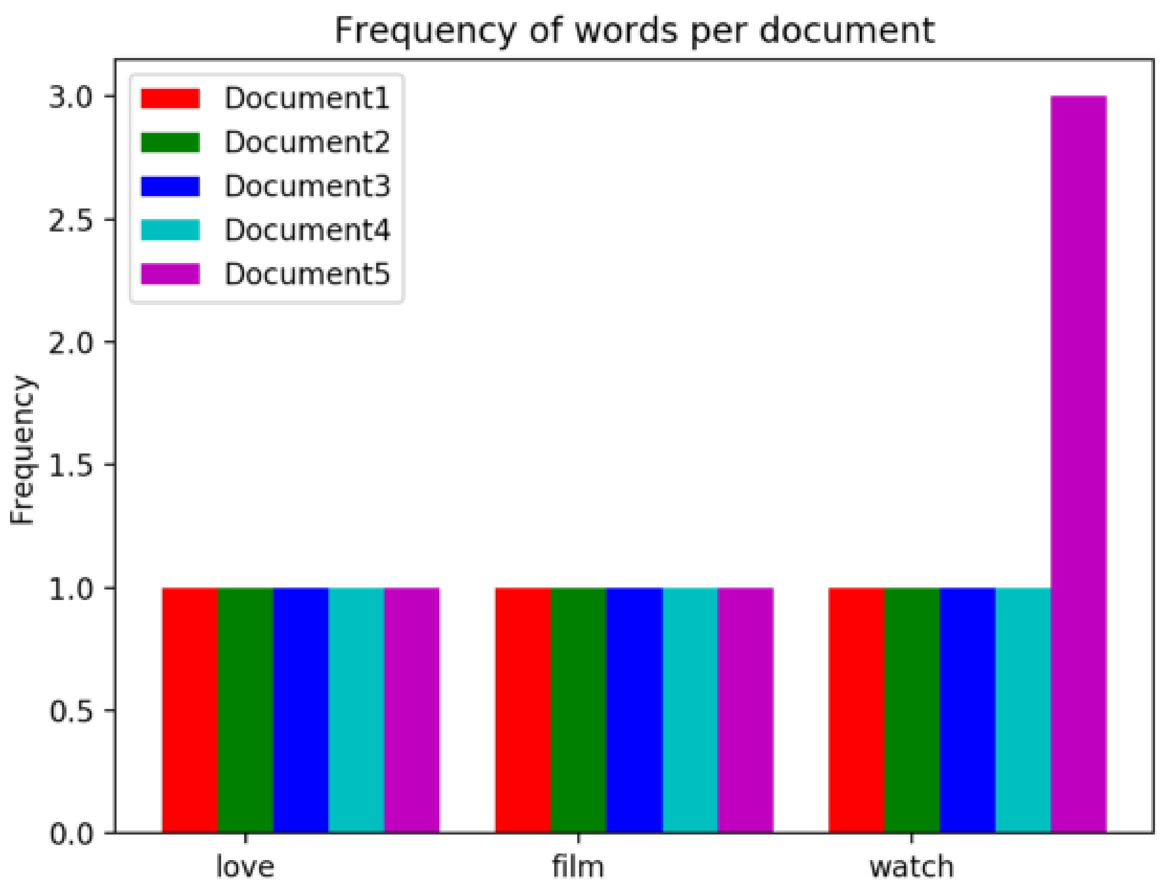


Wait…It seems there's something wrong with our histogram. The verb **love** appears in all documents, but here we can see it only in document 1 and document 2.

Actually, there's nothing wrong with the plotting function we've just defined. **The problem is what we are counting.** Computers can't tell that words like *love, loves, loving* and *loved* mean the same thing in essence. Thus, variations of a word are counted as different things.

The first part of our work is going to be removing prefixes and suffixes. That's what *stemming* does. Then, we'll go for lemmatization, which means we're going to remove inflections of all words in each document, such as '-s' or '-es' of plurals, '-ing' and '-ed' inflections.

If we plot a histogram of our collection again, we'll see that word distributions are now more consistent with our data.



Normalized Term Frequency

If we try to represent a search query and the talk transcripts by raw term frequency we will face a critical problem: there will be a bias for long documents.

A search query is much more shorter than a transcript, so that's not a option to us. To avoid this problem, we can calculate the normalized term frequency.

$$tf_{norm} = 1 + \log(tf_{raw})$$

In the same script, define a new function to calculate the normalized term frequency.

Inverse Document Frequency

The term frequency, wether normalized or not, doesn't tell much about how meaningful that term is. When it comes to a collection of documents of similar topic, some terms end up not carrying much information despite their high frequency.

For example, let's say we have a collection of talks about ***diet,*** each of them describing one different type of diet. The word ***food*** is likely to appear many times in all talks, but it won't give us much information about any of the diets. On the other hand, a word that doesn't appear much across all talks like ***detox*** can tell us directly what type of diet is being discussed in one of the talks.

The purpose of Inverse Document Frequency is exactly that. It evaluates the relevance of a term by measuring how often it appears in a collection of documents. It's high for words that carry a lot of information, and low for words that don't.

$$idf(t) = 1 + \log(\frac{n_{total}}{n_{docs(t)}})$$

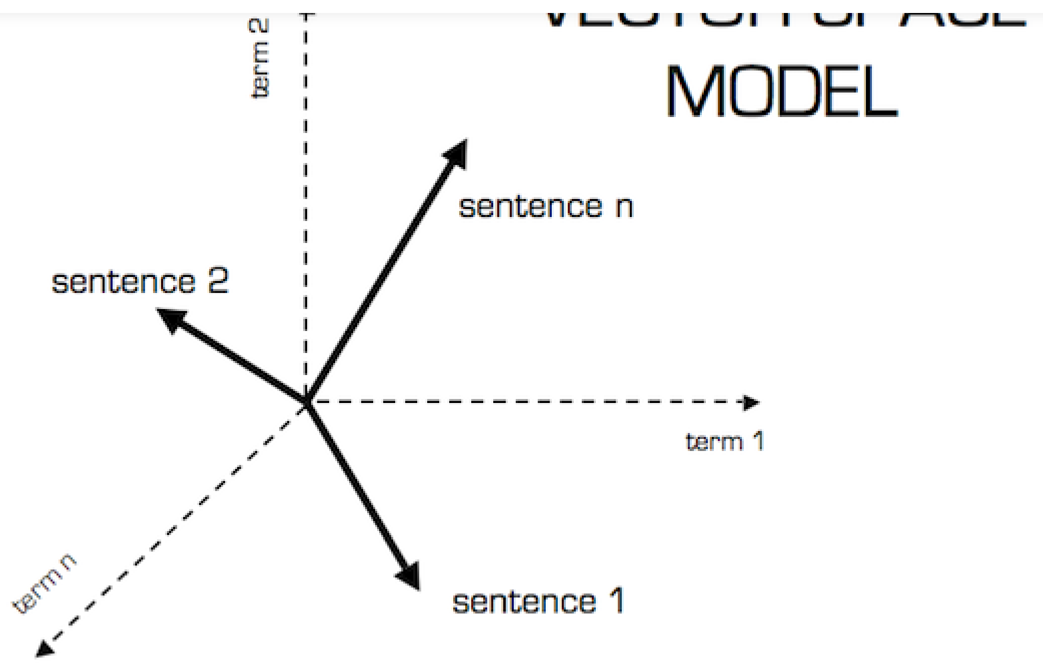
All in one

Now that we've understand how TF-IDF works, let's be more practical. Thanks to Scikit-Learn everything is pretty much done and we can calculate the TF-IDF matrix with just a few lines of code.

And no, all this time you've spent coding TF-IDF from scratch was not wasted. If you compare the TF-IDF matrix calculated with Scikit-Learn and the matrix calculated with your own version you'll see they are equal.

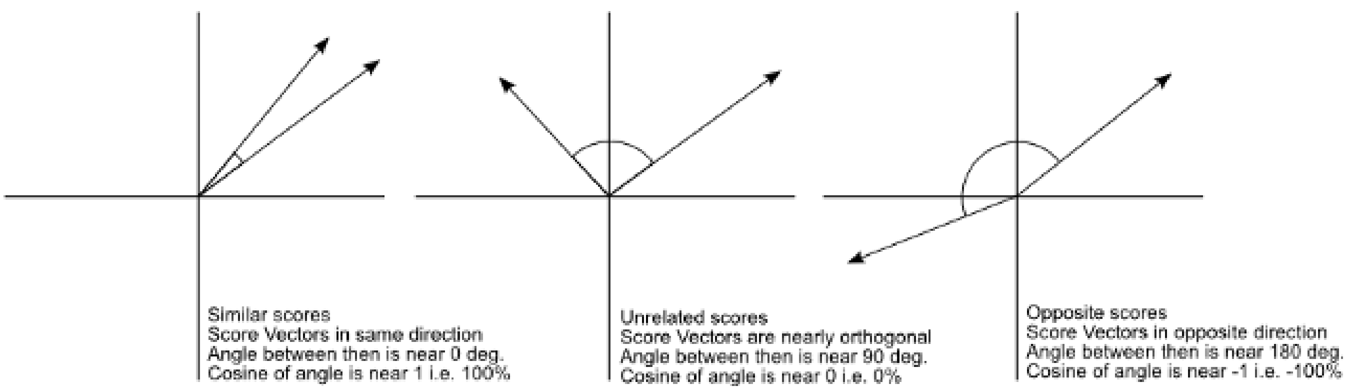
Cosine Similarity

Search queries and each document in our collection are now represented by a vector of TF-IDF weights. We need to find which documents are more relevant to the user. In other words, we have to find which document vectors are more close to the search query vector. There're many measures used to find the distance between vectors , like Euclidean Distance and Manhattan Distance. But what better fits our case is **Cosine distance**.



Representation of documents as vectors(taken from https://goo.gl/ppES7b)

Cosine distance measures the similarity between two vectors by calculating the cosine of the angle between them. It takes into account the orientation of vectors, not their magnitude. In this way, we not only consider the TF-IDF weights of each document(magnitude), but also the angle between them. This is extremely important since the magnitude of search query vectors will be much more lower than for the other documents.


Similar scores
Score Vectors in same direction
Angle between then is near 0 deg.
Cosine of angle is near 1 i.e. 100%
Unrelated scores
Score Vectors are nearly orthogonal
Angle between then is near 90 deg.
Cosine of angle is near 0 i.e. 0%
Opposite scores
Score Vectors in opposite direction
Angle between then is near 180 deg.
Cosine of angle is near -1 i.e. -100%

Example of cosine similarity measures(taken from https://goo.gl/ppES7b)

So, let's calculate the cosine similarity between a search query vector and all other vectors.

Now, let’s write a function to pick the talks with highest values.

The function above returns a list with indexes of the *N* talks more related to a search query. After that, all we have to do is to return the video url correspondent to each of these indexes.

It's not over yet!

I hope you've found the first part of this tutorial helpful. We learned what TF-IDF is and how it works and used it to code the core of **TEDFinder**.

In the next part, we're going to use Django to build a nice interface and put our search engine to work!

Thanks to Hugo Dzin